Metodología de la Programación II

Recursividad

Objetivos

- Entender el concepto de recursividad.
- Conocer los fundamentos del diseño de algoritmos recursivos.
- Comprender la ejecución de algoritmos recursivos.
- Aprender a realizar trazas de algoritmos recursivos.
- Comprender las ventajas e inconvenientes de la recursividad.

1.1 Concepto de Recursividad

La recursividad constituye una de las herramientas más potentes en programación. Es un concepto conocido. Por ejemplo,

- Definición recursiva

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

- Demostración por inducción: demostrar para un caso base y después para un tamaño `n`, considerando que está demostrado para menor que `n`.

Una función que se llama a sí misma se denomina **recursiva**

Podemos usar recursividad si la solución de un problema está expresada en términos de la resolución de un problema de la misma naturaleza, aunque de menor tamaño y conocemos la solución no-recursiva para un determinado caso.

- Ventajas: No es necesario definir la secuencia de pasos exacta para resolver el problema. Podemos considerar que *"tenemos resuelto"* el problema (de menor tamaño).
- Desventajas: Tenemos que encontrar una solución recursiva, y, además, podría ser menos eficiente.

Para que una definición recursiva esté completamente identificada es necesario tener un ***caso base*** que no se calcule utilizando casos anteriores y que la división del problema converja a ese caso base.

$$0! = 1$$

Ejemplo:

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ x \cdot x^{n-1} & \text{si } n > 0 \end{cases}$$

Ejemplo: Cálculo del factorial con n=3.

3! = 3 * 2!

(1)

3! = 3 * 2!
2! = 2 * 1!

(2)

3! = 3 * 2!
2! = 2 * 1!
1! = 1 * 0!

(3)

3! = 3 * 2!
2! = 2 * 1!
1! = 1 * 0!
0! = 1
(caso base)

(4)

3! = 3 * 2!
2! = 2 * 1!
1! = 1 * 1
1

(5)

3! = 3 * 2!
2! = 2 * 1
1! = 1 * 1 = 1

(6)

3! = 3 * 2
2! = 2 * 1 = 2

(7)

3! = 3 * 2 = 6

(8)

1.2 Diseño de algoritmos recursivos

El primer paso será la identificación de un algoritmo recursivo, es decir, descomponer el problema en subproblemas de menor tamaño (aunque de la misma naturaleza del problema original) y componer la solución final a partir de las subsoluciones obtenidas.

Diseñar:

1. casos base,
2. casos generales y
3. la solución, combinandos ambos.

- **Casos base**:

  Son los casos del problema que se resuelve con un segmento de código sin recursividad.

  Siempre debe existir al menos un caso base

  El número y forma de los casos base son hasta cierto punto arbitrarios. La solución será mejor cuanto más simple y eficiente resulte el conjunto de casos seleccionados.

- **Casos generales**:

  Si el problema es suficientemente complejo, la solución se expresa, de forma recursiva, como la unión de

1. La solución de uno o más subproblemas (de igual naturaleza pero menor tamaño).
2. Un conjunto de pasos adicionales. Estos pasos junto con las soluciones a los subproblemas componen la solución al problema general que queremos resolver.

Los casos generales siempre deben avanzar hacia un caso base. Es decir, la llamada recursiva se hace a un subproblema más pequeño y, en última instancia, los casos generales alcanzarán un caso base.

Ejemplo:

```
// Solucion no estructurada
int factorial (int n) {
  if (n==0) return (1); //Caso base
  else      return (n*factorial(n-1)); //Caso general
}

// Solucion estructurada
int factorial (int n) {
  int resultado;
  if (n==0) resultado = 1;   //Caso base
  else      resultado = n*factorial(n-1); //Caso general

  return (resultado);
}
```


1.3 Ejecución de un módulo recursivo

En general, en la pila se almacena el entorno asociado a las distintas funciones que se van activando.

En particular, en un módulo recursivo, cada llamada recursiva genera una nueva zona de memoria en la pila independiente del resto de llamadas.

Ejemplo: Ejecución del factorial con n=3.

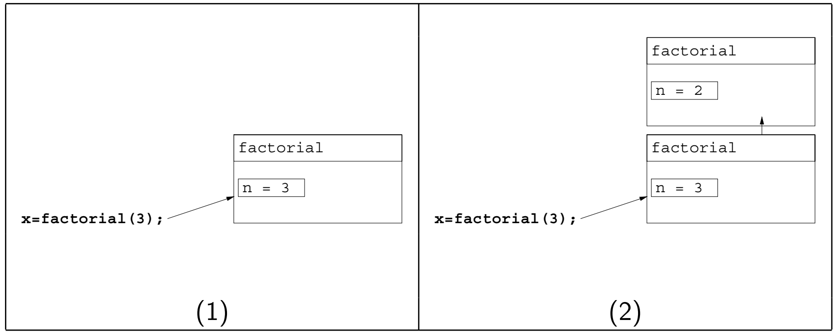
1. Dentro de factorial, cada llamada
   `return (n*factorial(n-1));`

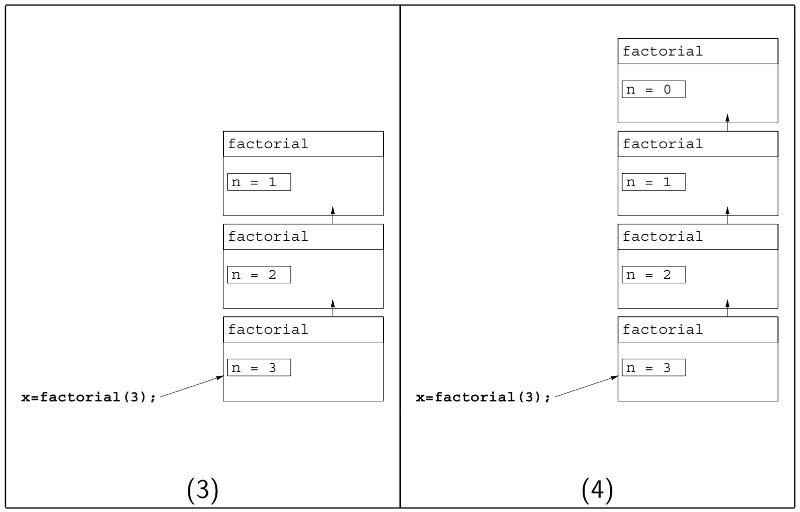
genera una nueva zona de memoria en la pila, siendo `n-1` el correspondiente parámetro actual para esta zona de memoria y queda pendiente la evaluación de la expresión y la ejecución del `return`.

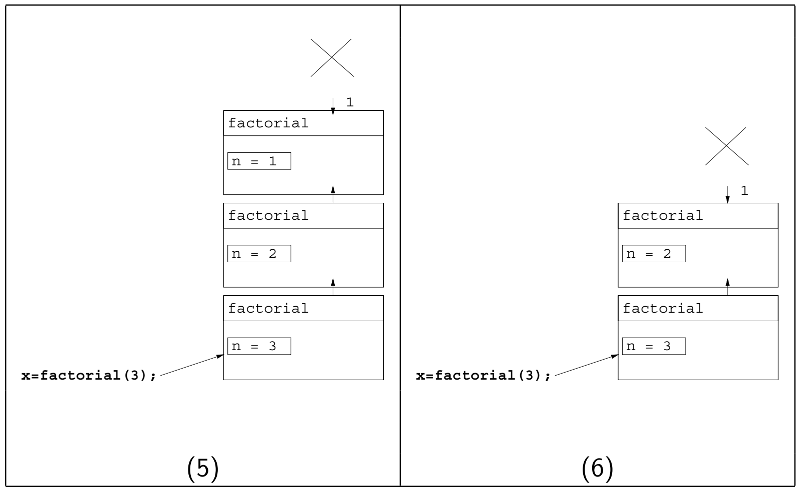
2. El proceso anterior se repite hasta que la condición del caso base se hace cierta.
   - Se ejecuta la sentencia `return (1);`
   - Empieza la vuelta atrás de la recursión, se evalúan las expresiones y se ejecutan los `return` que estaban pendientes.

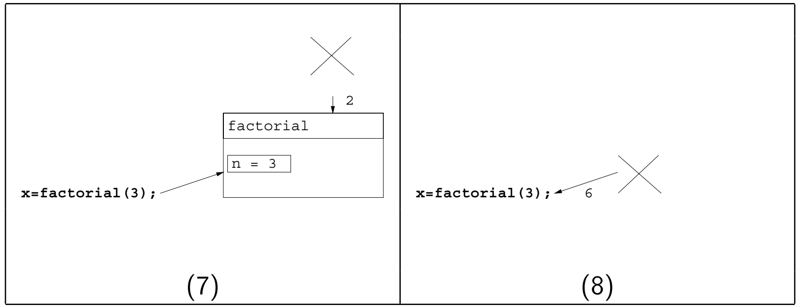
Gráficamente, la ejecución para la llamada:

```
x = factorial(3);
```



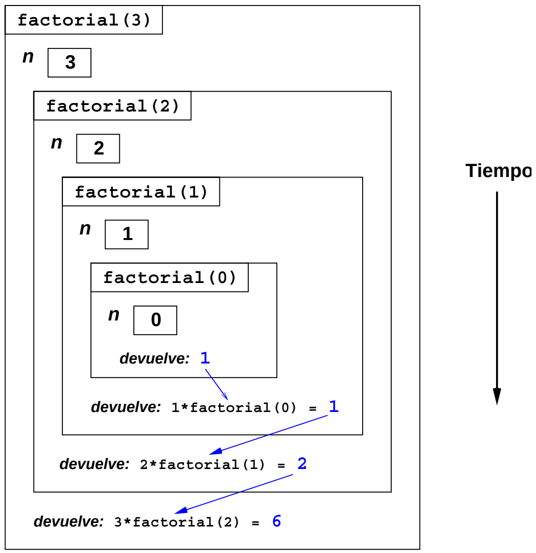




1.4 Traza de algoritmos recursivos

Se representan en cascada cada una de las llamadas al módulo recursivo, así como sus respectivas zonas de memoria y los valores que devuelven.

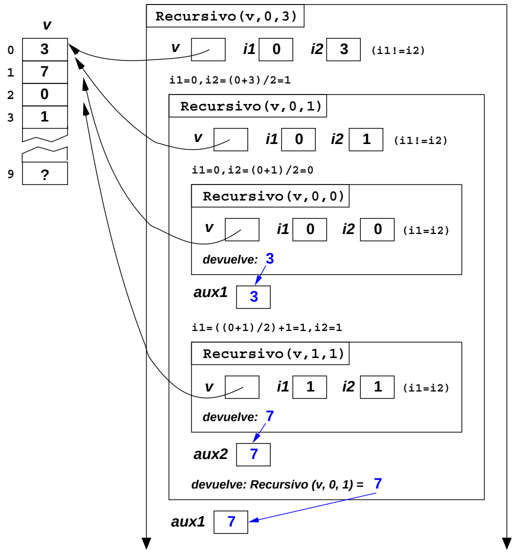
Llamada: `factorial(3)`

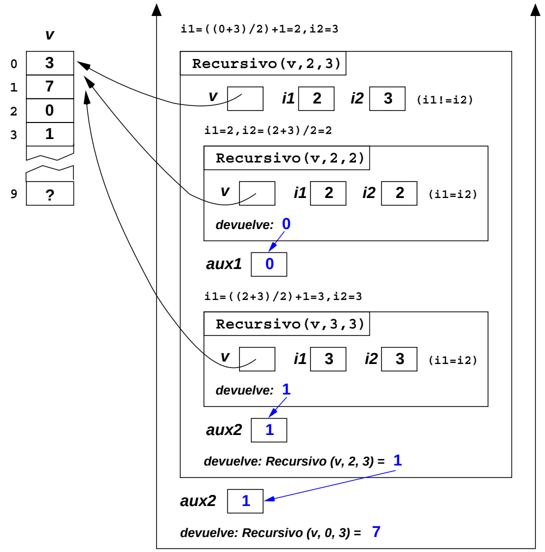


Ejemplo: Averiguar qué hace este algoritmo

```
int Recursivo (int *V, int i1, int i2)
{
   int aux1, aux2;

   if (i1==i2)  //Caso base
      return (V[i1]);
   else {       //Caso general
      aux1 = Recursivo(V, i1, (i1+i2)/2);
      aux2 = Recursivo(V, ((i1+i2)/2)+1, i2);
      return ((aux1> aux2) ? aux1 : aux2);
}
```

cuando se invoca: `Recursivo(V,0,3)`, siendo `V = [3,7,0,1]`





1.5 Ejemplos de funciones recursivas

1. Cálculo de la potencia

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ x \cdot x^{n-1} & \text{si } n > 0 \end{cases}$$

```
int potencia(int base, int expo)
{
  if (expo==0)
    return (1);
  else
    return (base * potencia(base,expo-1));
}
```


2. La suma de forma recursiva

$$suma(a,b) = \begin{cases} a & \text{si } b = 0 \\ 1 + suma(a, b-1) & \text{si } b > 0 \end{cases}$$

```
int suma(int a, int b)
{
  if (b==0)
    return (a);
  else
    return (1+suma(a,b-1));
}
```

3. El producto de forma recursiva

$$producto(a,b) = \begin{cases} 0 & \text{si } b = 0 \\ a + producto(a, b-1) & \text{si } b > 0 \end{cases}$$

```
int producto(int a, int b)
{
  if (b==0)
    return (0);
  else
    return (a+producto(a,b-1));
}
```


4. Suma recursiva de los elementos de un vector

$$sumaV(V,n) = \begin{cases} V[0] & \text{si } n = 0 \\ V[n] + sumaV(V, n-1) & \text{si } n > 0 \end{cases}$$

```
int SumaV (int *V, int n)
{
  if (n==0)
    return (V[0]);
  else
    return (V[n] + sumaV(V,n-1));
}
```


5. Buscar el máximo de un vector (I)

$$Mayor1(V,n) = \begin{cases} V[0] & \text{si } n = 0 \\ V[n] \text{ ó } Mayor1(V, n-1) & \text{si } n > 0 \end{cases}$$

```
int Mayor1 (int *V, int n)
{
  int aux;

  if (n==0)
    return (V[0]);
  else {
    aux = Mayor1 (V, n-1);
    return ((V[n]> aux) ? V[n] : aux);
}
```


6. Buscar el máximo entre dos posiciones de un vector

$$Mayor2(V,i,d) = \begin{cases} V[i] & \text{si } i = d \\ Mayor2(V, i, (i+d)/2 \text{ ó} & \\ Mayor2(V, ((i+d)/2) + 1, d) & \text{si } i < d \end{cases}$$

```
int Mayor2 (int *V, int izq, int der)
{
  int m_izq, m_der;
  if (izq==der) return (V[izq]);
  else {
    m_izq = Mayor2(V, izq, (izq+der)/2);
    m_der = Mayor2(V, ((izq+der)/2)+1, der);
    return ((m_izq> m_der) ? m_izq : m_der);
  }
}
```

7. Búsqueda lineal recursiva (con dos casos base)

Si n es el número de elementos del vector:

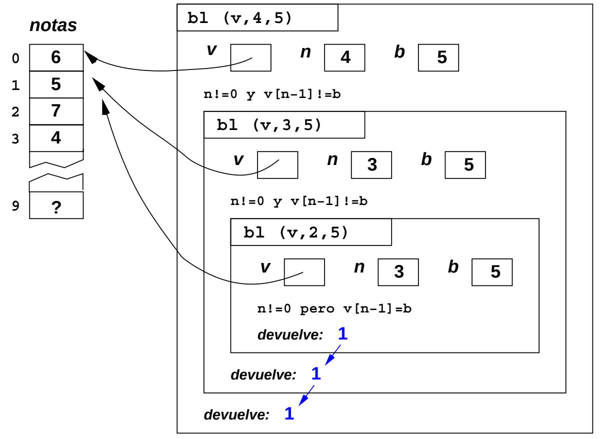
$$BusquedaLineal(V,n,b) = \begin{cases} \text{Verdad} & \text{si } V[n-1] = b \\ \text{Falso} & \text{si } V[0] \neq b \\ BusquedaLineal(V,n-1,b) & \text{en otro caso} \end{cases}$$

```
bool BusquedaLineal(int *V, int n, int b)
{
   if (n != 0) {
      if (V[n-1]==b) return (true);
      else return (BusquedaLineal(V,n-1,b));
   }
   else return (false);
 }
```


Ejemplo

Búqueda con éxito:

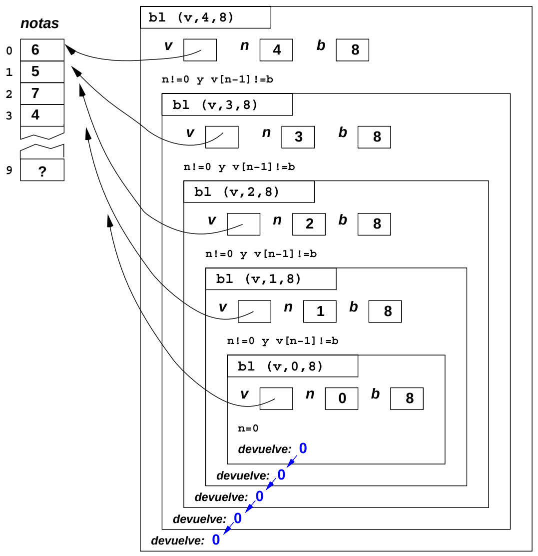
`BusquedaLineal(notas, 4, 5);`



Ejemplo

Búqueda con fracaso:

`BusquedaLineal(notas, 4, 8);`



1.6 Ejemplos avanzados

1.6.1 Busqueda binaria recursiva

- **Motivación**: La búsqueda en un *vector ordenado* se puede realizar comparando el valor buscado con el elemento central:
  - Si es igual, la búsqueda termina con éxito.
  - Si es menor, la búsqueda debe continuar en el subvector izquierdo.
  - Si es mayor, la búsqueda debe continuar en el subvector derecho.

- Cabecera de una función de búsqueda:

```
int BUSCA (int v[], int i, int d, int x);
```

Devuelve la posición en `v` donde se encuentra `x`.

La búsqueda se realiza entre las posiciones `i` y `d`. Si `x` no está en el vector, la función devuelve `-1`.

- Líneas básicas (`BUSCA (v, t+1, d,   x)`):

1. Seleccionar una casilla cualquiera, t, entre las casillas i y j ($i \leq t \leq j$). Sea $c = v[t]$. P.e. $t = (i + j)/2$
2. Comparar c con x.

   a) Si $c = x$, el elemento buscado está en la posición t (**Éxito**).

   b) Si $c < x$, el elemento buscado debe estar en una posición mayor que t: `BUSCA (v, t+1, d,   x)`

   c) Si $c > x$, el elemento buscado debe estar en una posición menor que t: `BUSCA (v, i,   t-1, x)`

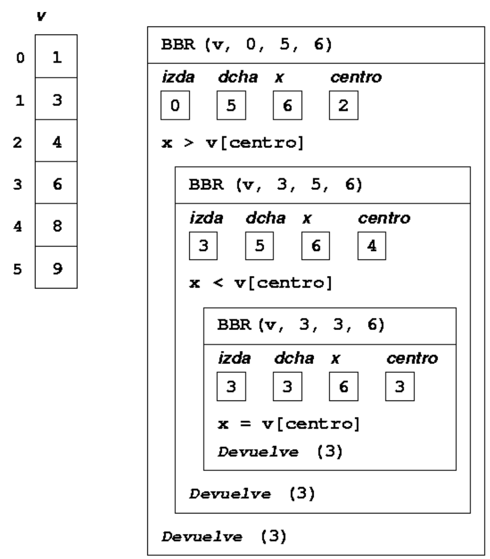
   d) Al modificar los extremos puede darse el caso de que $i > d$ (**Fracaso**).


```
int BBR (int v[], int i, int d, int x)
{
   int centro;

   if (i<=d)  {
      centro = (i+d)/2;

      if (v[centro]==x)   return (centro); // Caso base 1: EXITO
      else
         if (v[centro]>x) return (BBR (v,i,centro-1,x)); // Izda.
         else             return (BBR (v,centro+1,d,x)); // Dcha.
   }
   else // i > d
      return (-1); // Caso base 2 : FRACASO
}
```


Ejemplo

Búqueda con éxito:

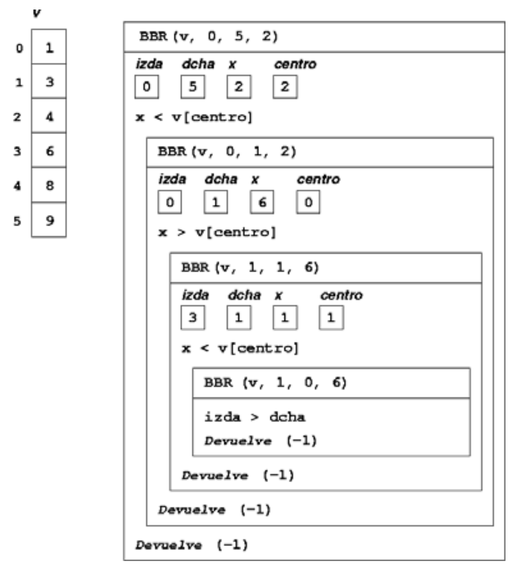
`BBR(v, 0, 5, 6);`



Ejemplo

Búqueda con fracaso:

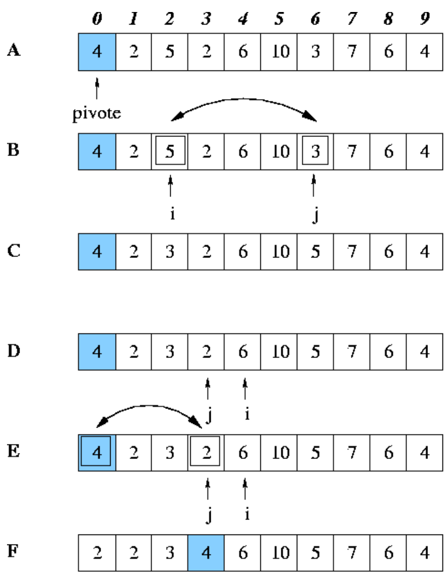
```
BBR(v, 0, 5, 2);
```



1.6.2 Ordenación rápida

- **Líneas básicas**:

1. Tomar un elemento arbitrario del vector: ***pivote***. Sea p su valor.
2. Recorrer el vector de izquierda a derecha hasta encontrar un elemento situado en una posición i tal que $v[i] > p$.
3. Recorrer el vector de derecha a izquierda hasta encontrar otro elemento situado en una posición j tal que $v[j] < p$.
4. Intercambiar los elementos de las casillas i y j (ahora, $v[i] < p < v[j]$).
5. Repetir hasta que los dos procesos de recorrido se encuentren ($i > j$).
6. Colocar el pivote en el sitio que le corresponde.
   El vector está particionado en dos zonas delimitadas por el pivote.




```
void OR (int *v, int izda, int dcha)
{
   int partir (int *v, int primero, int ultimo);  // Funcion de particion
   int pos_pivote; // Pos. del pivote tras particion

   if (izda < dcha) {

      // Particionar "v"
      pos_pivote = partir (v, izda, dcha);

      // Ordenar la primera mitad
      OR (v, izda, pos_pivote-1);

      // Ordenar la segunda mitad
      OR (v, pos_pivote+1, dcha);
   }
}
```

```
int partir (int *v, int primero, int ultimo)
{
   void swap_int (int &a, int &b);

   int izda, dcha;  // Indices para recorrer v
   int val_pivote = v[primero];   // El pivote es el primer elemento.

   izda = primero + 1;  // "izda" va a la dcha.
   dcha = ultimo;       // "dcha" va a la izda.

   do { // Buscar e intercambiar elementos

      // Buscar  mayor que el pivote desde la izquierda
      while ((izda<=dcha) && (v[izda]<=val_pivote)) izda++;

      // Buscar menor o igual que el pivote desde la derecha
      while ((izda<=dcha) && (v[dcha]>val_pivote))  dcha--;
```

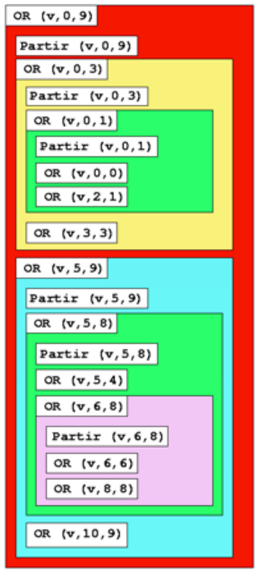


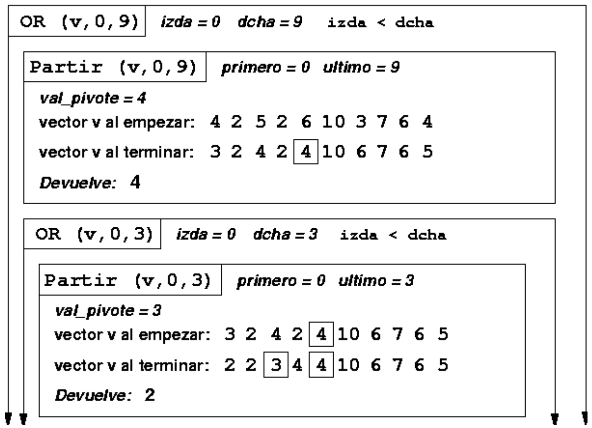
```
      if (izda < dcha) { // Intercambiar
         swap_int (v[izda], v[dcha]);
         dcha--;
         izda++;
      }

   } while (izda <= dcha); // Terminar cuando se cruzan "izda" y "dcha"

   // Colocar el pivote en su sitio correcto
   swap_int (v[primero], v[dcha]);

   return (dcha); // Devolver la pos. del pivote
}
```

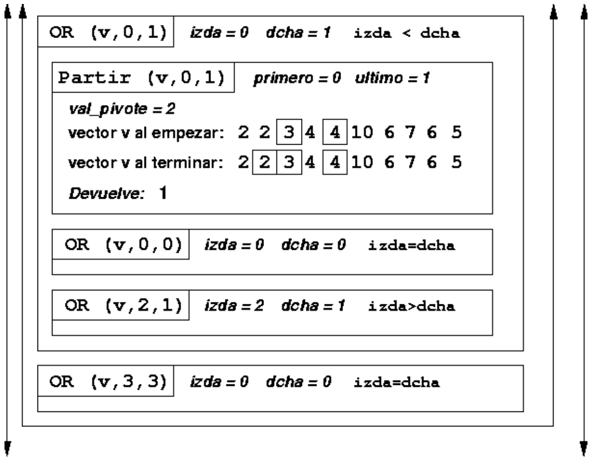

Ejemplo

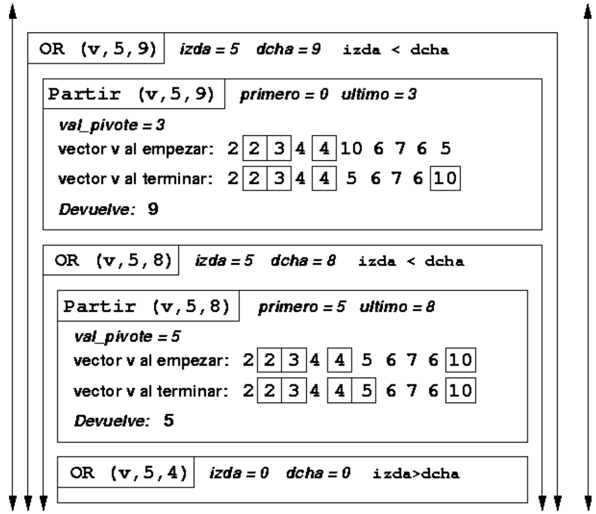
Resumen de:

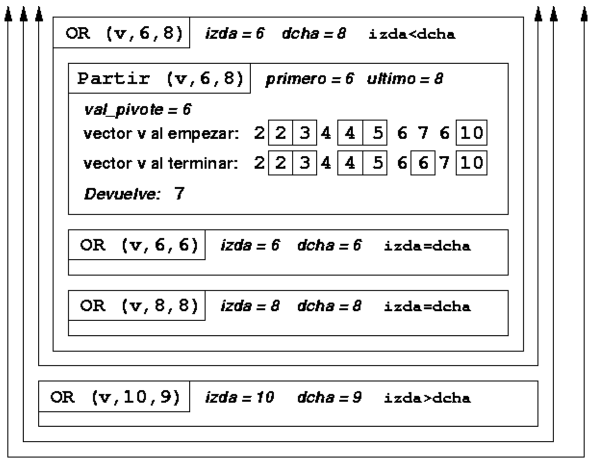
`OR (v, 0, 9);`











1.7 ¿Recursividad o iteración?

- Cuestiones a tener en cuenta:

1. La carga computacional (tiempo-espacio) asociada a una llamada a una función y el retorno a la función que hace la llamada.
2. Algunas soluciones recursivas pueden hacer que la solución para un determinado tamaño del problema se calcule varias veces.
3. Muchos problemas recursivos tienen como caso base un problema de un tamaño reducido. En ocasiones es *excesivamente* pequeño.
4. Puede ser muy difícil encontrar una solución iterativa eficiente.
5. La solución recursiva es muy concisa, legible y elegante.

1.7.1 Sucesión de Fibonacci

$$Fib(0) = Fib(1) = 1$$
$$Fib(n) = Fib(n-1) + Fib(n-2)$$

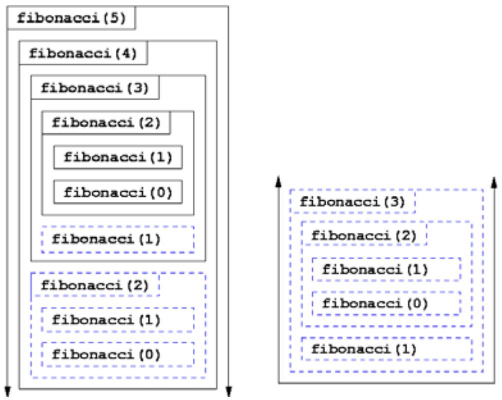
Solución recursiva:

```
/***********************************************/
int fibonacci (int n)
{
   if ((n == 0) || (n == 1)) return (1);
   else return (fibonacci(n-1) + fibonacci(n-2));
}
/***********************************************/
```


Solución iterativa:

```
/***********************************************/
int fibonacci (int n)
{
   int ant1 = 1, ant2 = 1; // anterior y anteanterior
   int actual;             // valor actual

   if ((n == 0) || (n == 1)) actual = 1;
   else
      for (i=2; i<=n; i++) {
         actual = ant1 + ant2; // suma los anteriores
         ant2 = ant1;          // actualiza "ant2"
         ant1 = actual;        // y "ant1"
      }
   return (actual);
}
/***********************************************/
```


Ejemplo: Cálculo de `fibonacci (5)`



1.7.2 Búsqueda binaria recursiva (2)

```
/********************************************/
int BBR2 (int v[], int i, int d, int x)
{
   int BLineal (int v[], int i, int d, int x);

   bool pequenio_BBR (int);

   if (pequenio_BBR (d-i+1))      // usar un algoritmo simple
      return (BLineal (v,i,d,x));

   else { // profundizar en la recursividad

      int centro = (i+d)/2;
```

```
      if (v[centro]==x) // Exito
         return (centro);

      else { // Seguir buscando

         if (v[centro]>x) // Buscar izda.
            return (BBR (v,i,centro-1,x));
         else // Buscar dcha.
            return (BBR (v,centro+1,d,x));
      }
   }
}
/*********************************************/
```



```
/*********************************************/
bool pequenio_BBR (int n)
{
   return (n <= BBR_UMBRAL);
}
/*********************************************/

int BLineal (int v[], int i, int d, int x)
{
   bool encontrado=false;

   for (int p=i; (i<d) && (!encontrado); i++)
       if (v[i] == x) encontrado = true;

   return ((encontrado) ? i : -1);
}
/*************************************************/
```


- Notas:

1. El caso base 2 (Fracaso) de la función `BBR()` ya no es necesario porque no se debe dar el caso de que $i > d$.
2. Es obligatorio que la función que resuelve el problema para un tamaño pequeño (`BLineal()`) devuelva un valor coherente con el que devuelve `BBR2()`.

1.7.3 Ordenación rápida (2)

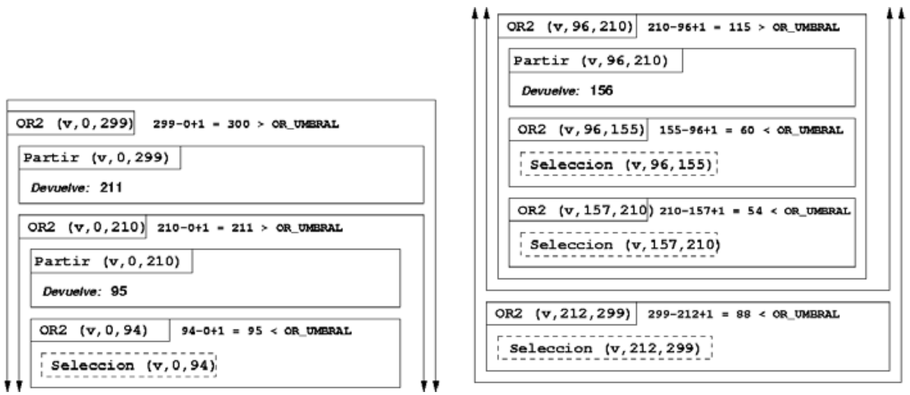
```
/**********************************************************/

void OR2 (int *v, int izda, int dcha)
{
   void seleccion (int *v, int izda, int dcha);
   int  partir (int *v, int primero, int ultimo);
   int  pequenio_OR (int n);

   if (pequenio_OR (dcha-izda+1)) // usar un algoritmo simple
      seleccion (v, izda, dcha);

   else {

      if (izda < dcha) {
          int pos_pivote; // Pos. pivote tras partir
```

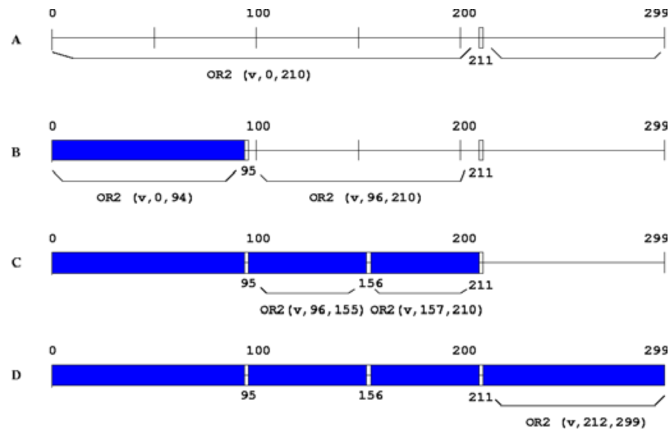
```
         // Particionar "v"
         pos_pivote = partir (v, izda, dcha);

         OR2 (v, izda, pos_pivote-1);  // Ordenar la primera mitad
         OR2 (v, pos_pivote+1, dcha);  // Ordenar la segunda mitad
      }
   }
}
/************************************************************/

bool pequenio_OR (int)
{
   return (n <= OR_UMBRAL);
}
/************************************************************/
```



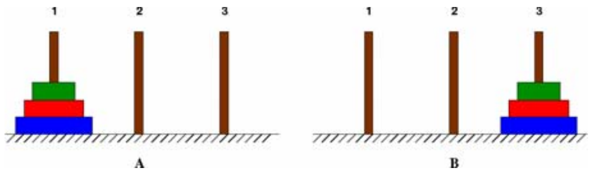
```
void seleccion (int *v, int izda, int dcha)
{
   int i, j, pos_menor;
   int menor;

   for (i = izda; i < dcha; i++) {
      pos_menor = i;
      menor = v[i];
      for (j = i+1; j <= dcha; j++)
         if (v[j] < menor) {
            pos_menor = j;
            menor = v[j];
         }
      v[pos_menor] = v[i];
      v[i] = menor;
   }
}
```


Ejemplo: `OR2 (v, 0, 299)` y `OR_UMBRAL=100`





1.7.4 Torres de Hanoi




```
#include <iostream>
using namespace std;

int main (void)
{
   void hanoi (int n, int inic, int tmp, int final);
   int n; // Numero de discos a mover

   cout << "Numero de discos: ";
   cin >> n;

   hanoi (n, 1, 2, 3); // mover "n" discos del 1 al 3,
                       // usando el 2 como temporal.
   return (0);
}
```



```
void hanoi (int n, int inic, int tmp, int final)
{
   if (n > 0) {

      // Mover n-1 discos de "inic" a "tmp".
      // El temporal es "final".
      hanoi (n-1, inic, final, tmp);

      // Mover el que queda en "inic"a "final"
      cout <<"Del poste "<<inic<<" al "<<final<<"\n";

      // Mover n-1 discos de "tmp" a "final".
      // El temporal es "inic".
      hanoi (n-1, tmp, inic, final);
    }
}
```



```
% hanoi
Numero de discos: 3
Del poste  1 al  3
Del poste  1 al  2
Del poste  3 al  2
Del poste  1 al  3
Del poste  2 al  1
Del poste  2 al  3
Del poste  1 al  3
```